10.1 Modular compilation

Using separate files to organize code can also help to make the compilation process more efficient. So far, we used a ***single-step compilation*** approach in which all source files are compiled at the same time to create the executable, e.g. `gcc main.c threenumsfcts.c`. This approach has a significant drawback. Anytime one of the source files is modified, all files must be recompiled. Even a small change, such as modifying a printf statement to fix a spelling error, would require all files be recompiled. The following animation illustrates the single-step compilation.

PARTICIPATION ACTIVITY 10.1.1: Single-step compilation process.

Animation captions:

1. Single-step compilation compiles and links all source files and libraries.
2. Change made to main.c requires that main.c be recompiled. But, single-step compilation will recompile all source files.

As programs become larger and the number of source files increases, the time required to recompile and link all source files can become very long - often requiring minutes to hours. Instead of compiling an executable using a single step, a ***modular compilation*** approach can be used that separates the compiling and linking steps within the compilation process. In this approach, each source file is independently compiled into an object file. An ***object file*** contains machine instructions for the compiled code along with placeholders, often referred to as references, for calls to functions or accesses to variables or classes defined in other source files or libraries.

For a program involving two files main.c and threenumsfcts.c, the files can be compiled separately into two object files named main.o and threenumsfcts.o respectively. The resulting object files will include several placeholders for functions that are defined in other files. For example, the main.o object file may contain placeholders for calls to any functions in threenumsfcts.o.

After each source file has been compiled, the ***linker*** will create the final executable by ***linking*** together the object files and libraries. For each placeholder found within an object file, the linker searches the other object files and libraries to find the referenced function or variable. When linking the main.o and threenumsfcts.o files, the placeholder for a call to a function in main.o will be replaced with a jump to the first instruction of that function within the threenumsfcts.o object file. This creates a link between the code within the main.o and threenumsfcts.o object files. Once all placeholders have been linked together, the final executable can be created. The following animation illustrates.

PARTICIPATION ACTIVITY 10.1.2: Modular compilation process.

Animation captions:

1. Modular compilation separates the compiling and linking. Source files are first compiled to object files.
2. Linking process resolves references in object files and libraries to create a final executable.
3. Linker will search object files and libraries to find the referenced functions or variables.
4. Once the object files have been linked together, the final executable is created.
5. Changes to main.c only require that main.c be recompiled. threeintsfcts.c does not need to be recompiled. Executable is re-linked using new main.o object file and exisiting threeintsfcts.o object file.

Using a modular compilation approach has the benefit of reducing the time required to recompile and link the program executable. Instead of recompiling all source files, only the source files that have been modified need to be recompiled to create an updated object file. The linker can then use these newly recompiled object files along with the previously compiled object files for any unmodified source files to create the updated executable.

PARTICIPATION ACTIVITY 10.1.3: Modular compilation.

1) A single-step compilation approach is faster than a modular compilation approach.
   - ○ True
   - ○ False

2) An object file contains machine instructions for the compiled code, along with references to functions or access to variables or classes in other files or libraries.
   - ○ True
   - ○ False

3) A linker is responsible for creating object files.
   - ○ True
   - ○ False

10.2 Separate files for structs

Programmers typically put all code for a struct into two files, separate from other code.

- ***StructName.h*** contains the struct definition, including data members and related function declarations.
- ***StructName.c*** contains related function definitions.

A file that uses the struct, such as a main file or StructName.c, must include StructName.h. The .h file's contents are sufficient to allow compilation, as long as the corresponding .c file is eventually compiled into the program too.

The figure below shows how all the .c files might be listed when compiled into one program. Note that the .h file is not listed in the compilation command, due to being included by the appropriate .c files.

Figure 10.2.1: Using two separate files for a struct.

StoreItem.h

```
#ifndef STOREITEM_H
#define STOREITEM_H

typedef struct StoreItem_struct {
   int weightOunces;
   // (other fields omitted for brevity)
} StoreItem;

void StoreItemSetWeightOunces
        (StoreItem* storeItem, int
weightOunces);
void StoreItemPrint(StoreItem
storeItem);

#endif
```

StoreItem.c

```
#include <stdio.h>
#include "StoreItem.h"

void StoreItemSetWeightOunces
        (StoreItem* storeItem, int weightOunces)
{
   (*storeItem).weightOunces = weightOunces;
}

void StoreItemPrint(StoreItem storeItem) {
   printf("Weight (ounces): %d\n",
storeItem.weightOunces);
}
```

main.c

```
#include <stdio.h>
#include "StoreItem.h"

int main() {
   StoreItem item1;

   StoreItemSetWeightOunces(&item1,
16);
   StoreItemPrint(item1);

   return 0;
}
```

Compilation example

```
% gcc -Wall -Wextra  -std=c99 -pedantic StoreItem.c
main.c
% a.out
Weight (ounces): 16
```

Good practice for .c and .h files

Sometimes multiple small related structs are grouped into a single file to avoid a proliferation of files. But for typical structs, good practice is to create a unique .c and .h file for each struct.

PARTICIPATION ACTIVITY 10.2.1: Separate files.

1) Commonly a struct definition and associated function definitions are placed in a .h file.
   - ○ True
   - ○ False

2) The .c file for a struct should #include the associated .h file.
   - ○ True
   - ○ False

3) A drawback of the separate file approach is longer compilation times.
   - ○ True
   - ○ False

CHALLENGE ACTIVITY 10.2.1: Enter the output of separate files.

347282.2052980.qx3zqy7

Start

Type the program's output

main.c | Movie.h | Movie.c

```
#include <stdio.h>
#include "Movie.h"

int main() {
   Movie film1;

   MovieSetRating(&film1, 1);
   MovieSetRelease(&film1, 2015);
   MovieRatingPrint(film1);

   return 0;
}
```


1

Check Try again

10.3 Makefiles

While modular compilation offers many advantages, manually compiling and linking a program with numerous source and header files can be challenging. For example, a change within a single header file will require recompiling all source files that include that header file. Keeping track of these dependencies to determine which files need to be recompiled is not trivial and can require considerable effort, if done manually. For large programs, programmers often utilize ***project management tools*** to automate the compilation and linking process. ***make*** is one project management tool that is commonly used on Unix and Linux computer systems.

The make utility uses a ***makefile*** to recompile and link a program whenever changes are made to the source or header files. The makefile consists of a set of rules and commands. ***Make rules*** are used to specify dependencies between a target file (e.g., object files and executable) and a set of prerequisite files that are needed to generate the target file (e.g., source and header files). A make rule can include one or more commands -- referred to as ***make recipe*** -- that will be executed in order to generate the target file. The following shows the general form for a make rule. The make rule's target and prerequisites are defined on a single line. The commands for the make rule should be specified one per line starting with a single tab character.

Construct 10.3.1: Makefile rules and commands.

```
target : prerequisite1 prerequisite2 ... prerequisiteN
    command1
    command2
    ...
    commandN
```

A common error is to use spaces instead of a single tab character for specifying the commands. Another common error is including a tab character on an empty line, which will be reported by make as an error. As distinguishing between tabs and spaces in some text editors can be difficult, these errors are often challenging to find.

The following provides an example makefile for a program consisting of three files main.c, threeintsfcts.c, and threeintsfcts.h. The executable named myprog.exe is dependent on the object files main.o and threeintsfcts.o. The main.o object file is dependent on the main.c source file and the threeintsfcts.h header file. The threeintsfcts.o object file is dependent on the threeintsfcts.c and threeintsfcts.h files.

Figure 10.3.1: Makefile example for three integer functions program.

```
myprog.exe : main.o threeintsfcts.o
    gcc main.o threeintsfcts.o -o myprog.exe

main.o : main.c  threeintsfcts.h
    gcc -Wall -c main.c

threeintsfcts.o : threeintsfcts.c threeintsfcts.h
    gcc -Wall -c threeintsfcts.c

clean :
    rm *.o myprog.exe
```

Running make for first time:	`> make`
	`gcc -Wall -c main.c`
	`gcc -Wall -c threeintsfcts.c`
	`gcc main.o threeintsfcts.o -o myprog.exe`

Running make after modifying main.c:	`> make`
	`gcc -Wall -c main.c`
	`gcc main.o threeintsfcts.o -o myprog.exe`

When the make command is executed, if any of the prerequisites for the rule have been modified since the target was last created, the commands for the rule will be executed to create the target file. For example, in the above makefile, if main.c is modified make will first execute the command gcc -Wall -c main.c to create the main.o object file. The -c flag is used here to inform the compiler (e.g. gcc) that the source file should only be compiled (and not linked) to create an object file. As main.o has

now been modified, make will then execute the command `gcc main.o threeintsfcts.o -o myprog.exe`. The -o flag is used here to inform the linker (e.g. gcc) to link the object files into the final executable using the name specified after the -o. In this case, the executable is named myprog.exe.

Make rules can also be used to define common operations used when managing larger programs. One common make rule is the `clean : rule` that is used to execute a command for deleting all generated files such as object files and the program executable. In the above example, this is accomplished by running the command `rm *.o myprog.exe`.

By default make assumes the makefile is named `makefile` or `Makefile`. The -f flag can be used to run make using a different filename. For example, `make -f MyMakefile` will run make using the file named `MyMakefile`.

PARTICIPATION ACTIVITY 10.3.1: Makefiles.

Answer the following using the Makefile provided above.

1) List the targets affected if main.c is changed. (Note: List the targets separated by spaces and in the order that their rules would execute.)

Check Show answer

2) What command will execute after threeintsfcts.o is changed? (Hint: The answer starts with gcc.)

Check Show answer

3) How many commands will execute when threeintsfcts.h is changed?

Check Show answer

Exploring further:

- Makefile tutorial from cprogramming.com
- Manual for make from gnu.org

10.4 The #define directive

The ***#define*** directive, of the form #define MACROIDENTIFIER replacement, instructs the processor to replace any occurrence of MACROIDENTIFIER in the subsequent program code by the replacement text.

Construct 10.4.1: #define directive.

```
#define MACROIDENTIFIER replacement
```

#define is sometimes called a ***macro***. The #define line does *not* end with a semicolon.

Most uses of #define are strongly discouraged by experienced programmers. However, for legacy reasons, #define appears in much existing code, so a programmer still benefits from understanding #define.

One (discouraged) use of #define is for a constant. The directive `#define MAXNUM 99` causes the preprocessor to replace every occurrence of identifier MAXNUM by 99, before continuing with compilation. So `if (x < MAXNUM)` will be replaced by if `(x < 99)`. In contrast, declaring a constant variable `const int MAXNUM = 99` has several advantages over a macro, such as type checking, syntax errors for certain incorrect usages, and more.

Another (discouraged) use of #define is for a type-neutral function. #define may specify arguments, as in `#define FCT1(a, b) ((a + b)/(a * b))`. A program may then have statements like `numInt = FCT1(1,2)` or like `numFloat = FCT1(1.2, 0.7)`, which the preprocessor would replace by `numInt = ((1 + 2) / (1 * 2))` or `numFloat = ((1.2 + 0.7) / (1.2 * 0.7))`, respectively. However, defining functions for each type is better practice.

Some advanced techniques make good use of macros, and are mentioned in other sections, such as a section introducing the assert macro.

PARTICIPATION ACTIVITY 10.4.1: #define.

Given:

```
#define PI_CONST  3.14159

double CalcCircleArea(double radius) {
   return PI_CONST * radius * radius;
}
```

1) The function call CalcCircleArea(1.0) returns 3.14159.

   - True
   - False

2) Replacing the return expression by `PI_CONST * PI_CONST * radius` yields a compiler error since only one replacement is allowed.

   - True
   - False

3) Replacing the return expression by `PI_CONSTPI_CONST` would yield 3.141593.14159, which would thus yield a compiler error complaining about the two decimal points.

   - True
   - False

4) The call CalcCircleArea(PI_CONST) would yield a compiler error complaining that the macro cannot be an argument.

   - True
   - False

5) Placing a semicolon at the end of the #define line will yield a compiler error at that line.

   - True
   - False

Exploring further:

- Other preprocessor directives include ***#undef***, ***#ifdef***, ***#if***, ***#else***, ***#elif***, ***#pragma***, ***#line***, and ***#error***.
- Preprocessor tutorial on cplusplus.com
- Preprocessor directives on MSDN

10.5 Engineering examples

This section has been set as optional by your instructor.

Arrays can be useful in solving various engineering problems. One problem is computing the voltage drop across a series of resistors. If the total voltage across the resistors is V, then the current through the resistors will be I = V/R, where R is the sum of the resistances. The voltage drop Vx across resistor x is then Vx = I · Rx. The following program uses an array to store a user-entered set of resistance values, computes I, then computes the voltage drop across each resistor and stores each in another array, and finally prints the results.

Figure 10.5.1: Calculate voltage drops across series of resistors.

```
5 resistors are in series.
Program calculates voltage drop across each
resistor.
Input voltage applied to circuit: 12
Input ohms of 5 resistors:
1) 3.3
2) 1.5
3) 2
4) 4
5) 2.2

Voltage drop per resistor is:
1) 3.0 V
2) 1.4 V
3) 1.8 V
4) 3.7 V
5) 2.0 V
```

```c
#include <stdio.h>

int main(void) {
   const int NUM_RES = 5;      // Number of
resistors
   double resVals[NUM_RES];    // Ohms
   double circVolt;            // Volts
   double vDrop[NUM_RES];      // Volts
   double currentVal;          // Amps
   double sumRes;              // Ohms
   int i;                      // Loop index

   printf("5 resistors are in series.\n");
   printf("Program calculates voltage drop ");
   printf("across each resistor.\n");

   printf("Input voltage applied to circuit:
");
   scanf("%lf", &circVolt);

   printf("Input ohms of %d resistors:\n",
NUM_RES);
   for (i = 0; i < NUM_RES; ++i) {
      printf("%d) ", i+1);
      scanf("%lf", &resVals[i]);
   }

   // Calculate current
   sumRes = 0;
   for (i = 0; i < NUM_RES; ++i) {
      sumRes = sumRes + resVals[i];
   }
   currentVal = circVolt / sumRes;    // I =
V/R

   for (i = 0; i < NUM_RES; ++i) {
      vDrop[i] = currentVal * resVals[i]; // V
= IR
   }

   printf("\nVoltage drop per resistor
is:\n");
   for (i = 0; i < NUM_RES; ++i) {
      printf("%d) %.1lf V\n", i + 1,
vDrop[i]);
   }

   return 0;
}
```


©zyBooks 12/22/21 12:35 1026490
Jacob Adams
UACS100Fall2021

PARTICIPATION ACTIVITY 10.5.1: Voltage drop program.

1) What does variable circVolt store?

- Multiple voltages, one for each resistor.

- The resistance of each resistor.
- The total voltage across the series of resistors.

2) What does the first for loop do?

- Gets the voltage of each resistor and stores each in an array.
- Gets the resistance of each resistor and stores each in an array.
- Adds the resistances into a total value.

3) What does the second for loop do?

- Adds the resistances into a single value, so that I = V/R can be computed.
- Computes the voltage across each resistor.

4) What does the third for loop do?

- Update the resistances array with new resistor values.
- Sum the voltages across each resistor into a total voltage.
- Determines the voltage drop across each resistor and stores each voltage in another array.

5) Could the fourth loop's statement have been incorporated into the third loop, thus eliminating the fourth loop?

- No, a resistor's voltage drop isn't known until the entire loop has finished.
- Yes, but keeping the loops separate is better style.

Engineering problems commonly involve matrix representation and manipulation. A matrix can be captured using a two-dimensional array. Then matrix operations can be defined on such arrays. The

following illustrates matrix multiplication for 4x2 and 2x3 matrices captured as two-dimensional arrays.

Figure 10.5.2: Matrix multiplication of 4x2 and 2x3 matrices.

```
#include <stdio.h>

int main(void) {
   const int M1_ROWS = 4;       // Matrix 1 rows
   const int M1_COLS = 2;       // Matrix 1 cols
   const int M2_ROWS = M1_COLS; // Matrix 2 rows (must have same value)
   const int M2_COLS = 3;       // Matrix 2 cols
   int i;                       // Loop index
   int j;                       // Loop index
   int k;                       // Loop index
   int dotProd;                 // Dot product

   // Populate matrices
   int m1[4][2] = {{3, 4},
                   {2, 3},
                   {1, 5},
                   {0, 2}};

   int m2[2][3] = {{5, 4, 4},
                   {0, 2, 3}};

   int m3[4][3] = {{0, 0, 0},
                   {0, 0, 0},
                   {0, 0, 0},
                   {0, 0, 0}};

   // m1 * m2 = m3
   for (i = 0; i < M1_ROWS; ++i) {
      for (j = 0; j < M2_COLS; ++j) {
         // Compute dot product
         dotProd = 0;
         for (k = 0; k < M2_ROWS; ++k) {
            dotProd = dotProd + (m1[i][k] * m2[k][j]);
         }

         m3[i][j] = dotProd;
      }
   }

   // Print m3 result
   for (i = 0; i < M1_ROWS; ++i) {
      for (j = 0; j < M2_COLS; ++j) {
         printf("%2d ", m3[i][j]);
      }
      printf("\n");
   }

   return 0;
}
```



```
15 20 24
10 14 17
 5 14 19
 0  4  6
```


PARTICIPATION ACTIVITY 10.5.2: Matrix multiplication program.

1) For the first set of for loops, how many dot products are computed? (In other words, how many iterations are due to the outer two for loops?)

Check Show answer

2) For the first set of for loops, the inner-most loop computes a dot product. Each time the inner-most loop is reached, how many times will the inner-most loop iterate?

Check Show answer

10.6 Engineering examples using functions

This section has been set as optional by your instructor.

This section contains examples of functions for various engineering calculations.

Gas equation

An equation used in physics and chemistry that relates pressure, volume, and temperature of a gas is PV = nRT. P is the pressure, V the volume, T the temperature, n the number of moles, and R a constant. The function below outputs the temperature of a gas given the other values.

Figure 10.6.1: PV = nRT. Compute the temperature of a gas.

```
Enter pressure (in Pascals):
2500
Enter volume (in cubic meters):
35.5
Enter number of moles: 18
Temperature = 593.01 K
```

```c
#include <stdio.h>

const double GAS_CONSTANT = 8.3144621;  //  J / (mol * K)

/* Converts a pressure, volume, and number of moles
 of a gas into a temperature. */
double PVnToTemp(double gasPressure, double gasVolume, double
numMoles) {
   return (gasPressure * gasVolume) / (numMoles *
GAS_CONSTANT);
}

int main(void) {
   double gasPress;  // User defined pressure
   double gasVol;  // User defined volume
   double gasMoles;  // User defined moles

   // Prompt user for input parameters
   printf("\nEnter pressure (in Pascals): ");
   scanf("%lf", &gasPress);

   printf("Enter volume (in cubic meters): ");
   scanf("%lf", &gasVol);

   printf("Enter number of moles: ");
   scanf("%lf", &gasMoles);

   // Call function to calculate temperature
   printf("Temperature = %.2lf K\n",
          PVnToTemp(gasPress, gasVol, gasMoles));

   return 0;
}
```


PARTICIPATION ACTIVITY 10.6.1: PV = nRT calculation.

Questions refer to PVnToTemp() above.

1) PVnToTemp() uses a rewritten form of PV = nRT to solve for T, namely T = PV/nR.

   - True
   - False

2) PVnToTemp() uses a constant variable for the gas constant R.

   - True
   - False

3)

TempVolMolesToPressure() would likely return (temp * vlm) / (mols * GAS_CONSTANT).

- True
- False

Projectile location

Common physics equations determine the x and y coordinates of a projectile object at any time, given the object's initial velocity and angle at time 0 with initial position x = 0 and y = 0. The equation for x is v * t * cos(a). The equation for y is v * t * sin(a) - 0.5 * g * t * t. The following provides a single function to compute an object's position; because position consists of two values (x and y), the function uses pass by pointer parameters to return values for x and y. The program's main function asks the user for the object's initial velocity, angle, and height (y position), and then prints the object's position for every second until the object's y position is no longer greater than 0 (meaning the object fell back to earth).

Figure 10.6.2: Trajectory of object on Earth.

```
Launch angle (deg): 45
Launch velocity (m/s): 100
Initial height (m): 3
Time   1   x =   0   y =   3
Time   2   x =  71   y =  66
Time   3   x = 141   y = 122
Time   4   x = 212   y = 168
Time   5   x = 283   y = 204
Time   6   x = 354   y = 231
Time   7   x = 424   y = 248
Time   8   x = 495   y = 255
Time   9   x = 566   y = 252
Time  10   x = 636   y = 239
Time  11   x = 707   y = 217
Time  12   x = 778   y = 185
Time  13   x = 849   y = 143
Time  14   x = 919   y =  91
Time  15   x = 990   y =  30
```

```
#include <stdio.h>
#include <math.h>

// Note: 1-letter variable names are typically avoided,
// but used below where standard in physics.

const double PI_CONST = 3.14159265;

// Given time, angle, velocity, and gravity
// Update x and y values
void ObjectTrajectory(double t, double a, double v, double g,
                      double* x, double* y) {
   *x = v * t * cos(a);
   *y = v * t * sin(a) - 0.5 * g * t * t;
}

// Convert degree value to radians
double DegToRad(double inDeg) {
   return ((inDeg * PI_CONST) / 180.0);
}

int main(void) {
   const double GRAVITY  = 9.8; // Earth gravity (m/s^2)
   double launchAngle;    // Angle of launch (rad)
   double launchVelocity; // Velocity (m/s)
   double elapsedTime;    // Time (s)

   double yLoc;          // Object's height above ground (m)
   double xLoc;           // Object's horiz. dist. from start (m)

   elapsedTime = 1.0;
   xLoc = 0.0;

   printf("Launch angle (deg): ");
   scanf("%lf", &launchAngle);
   launchAngle = DegToRad(launchAngle); // To radians

   printf("Launch velocity (m/s): ");
   scanf("%lf", &launchVelocity);

   printf("Initial height (m): ");
   scanf("%lf", &yLoc);

   while ( yLoc > 0.0 ) { // While above ground
      printf("Time %3.0f   x = %3.0lf   y = %3.0lf\n",
            elapsedTime, xLoc, yLoc);
      ObjectTrajectory(elapsedTime, launchAngle, launchVelocity,
                       GRAVITY, &xLoc, &yLoc);
      elapsedTime = elapsedTime + 1.0;
   }

   return 0;
}
```


©zyBooks 12/22/21 12:35 1026490
Jacob Adams
UACS100Fall2021

PARTICIPATION ACTIVITY 10.6.2: Projectile location.

Questions refer to ObjectTrajectory() above.

1) ObjectTrajectory() cannot return two values (for x and y), so instead takes x and y as modifiable parameters and changes their values.

- True
- False

2) ObjectTrajectory() could replace double types by int types without causing much change in computed values.

- True
- False

3) Each iteration of the loop will see yLoc increase.

- True
- False

4) Assuming the launch angle is less than 90 degrees, each iteration of the loop will see xLoc increase.

- True
- False

CHALLENGE ACTIVITY 10.6.1: Function to compute gas volume.

Define a function ComputeGasVolume that returns the volume of a gas given parameters pressure, temperature, and moles. Use the gas equation PV = nRT, where P is pressure in Pascals, V is volume in cubic meters, n is number of moles, R is the gas constant 8.3144621 (J / (mol*K)), and T is temperature in Kelvin.

347282.2052980.qx3zqy7


```
1  #include <stdio.h>
2
3  const double GAS_CONST = 8.3144621;
4
5  /* Your solution goes here  */
6
7  int main(void) {
8     double gasPressure;
9     double gasMoles;
10    double gasTemperature;
11    double gasVolume;
12
```

```
12
13    scanf("%lf", &gasPressure);
14    scanf("%lf", &gasMoles);
15    scanf("%lf", &gasTemperature);
16
17    gasVolume = ComputeGasVolume(gasPressure, gasTemperature, gasMoles);
18    printf("Gas volume: %lf m^3\n", gasVolume);
```

Run

10.7 Additional practice: Output art

This section has been set as optional by your instructor.

The following is a sample programming lab activity; not all classes using a zyBook require students to fully complete this activity. No auto-checking is performed. Users planning to fully complete this program may consider first developing their code in a separate programming environment.

The following program prints a simple triangle.

zyDE 10.7.1: Create ASCII art.

Load default template...

```
1  #include <stdio.h>
2
3  int main(void) {
4
5     printf("  *  \n");
6     printf(" *** \n");
7     printf("*****\n");
8
9     return 0;
10 }
11
```

Pre-enter any input for program, then run.

Run

Create different versions of the program:

1. Print a tree by adding a base under a 4-level triangle:

```
   *
  ***
 *****
*******
  ***
```


2. Print the following "cat":

```
^   ^
 o o
=   =
 ---
```

3. Allow a user to enter a number, and then print the original triangle using that number instead of asterisks, as in:

```
   9
  999
99999
```

Pictures made from keyboard characters are known as ***ASCII art***. ASCII art can be quite intricate, and fun to make and view. ASCII Art Archive provides examples. Doing a web search for "ASCII art (someitem)" can find ASCII art versions of an item. For example, searching for "ASCII art cat" turns up thousands of examples of cats, most much more clever than the cat above.

10.8 Additional practice: Grade calculation

This section has been set as optional by your instructor.

The following is a sample programming lab activity; not all classes using a zyBook require students to fully complete this activity. No auto-checking is performed. Users planning to fully complete this program may consider first developing their code in a separate programming environment.

zyDE 10.8.1: Grade calculator.

The following incomplete program should compute a student's total course percentage on scores on three items of different weights (%s):

- 20% Homeworks (out of 80 points)
- 30% Midterm exam (out of 40 points)
- 50% Final exam (out of 70 points)

Suggested (incremental) steps to finish the program:

1. First run it.
2. Next, complete the midterm exam calculation and run the program again. Use the constant variables where appropriate.
3. Then, complete the final exam calculation and run the program. Use the constant variables where appropriate.
4. Modify the program to include a quiz score out of 20 points. New weights: 10% homework, 15% quizzes, 30% midterm, 45% final. Run the program again.
5. To avoid having one large expression, introduce variables homeworkPart, quizPart midtermPart, and finalPart. Compute each part first; each will be a number betwe and 1. Then combine the parts using the weights into the course value. Run the pr again.

Load default templ

```
#include <stdio.h>

int main(void) {
   const double HOMEWORK_MAX = 80.0;
   const double MIDTERM_MAX = 40.0;
   const double FINAL_MAX = 70.0;
   const double HOMEWORK_WEIGHT = 0.20; // 20%
   const double MIDTERM_WEIGHT = 0.30;
   const double FINAL_WEIGHT = 0.50;

   double homeworkScore;
   double midtermScore;
   double finalScore;
   double coursePercentage;

   printf("Enter homework score:\n");
   scanf ("%lf", &homeworkScore);

```


78 36 62

Run

10.9 Example: Health data

This section has been set as optional by your instructor.

Calculating user's age in days

The section presents an example program that computes various health related data based on a user's age using incremental development. ***Incremental development*** is the process of writing, compiling, and testing a small amount of code, then writing, compiling, and testing a small amount more (an incremental amount), and so on.

The initial program below calculates a user's age in days based on the user's age in years. The assignment statement `userAgeDays = userAgeYears * 365;` assigns userAgeDays with the product of the user's age and 365, which does not take into account leap years.

Figure 10.9.1: Health data: Calculating user's age in days.

```
#include <stdio.h>

int main(void) {
   int userAgeYears;
   int userAgeDays;

   printf("Enter your age in years: ");
   scanf("%d", &userAgeYears);

   userAgeDays = userAgeYears * 365;

   printf("You are %d days old.\n", userAgeDays);

   return 0;
}
```

```
Enter your age in years: 19
You are 6935 days old.
```


PARTICIPATION ACTIVITY 10.9.1: Calculating user age in days.

1) Which variable is used for the user's age in years?

Check Show answer

2) If the user enters 10, what will userAgeYears be assigned?

Check Show answer

3) If the user enters 10, what is userAgeDays assigned?

Check Show answer

Considering leap years and calculating age in minutes

The program below extends the previous program by accounting for leap years when calculating the user's age in days. Since each leap year has one extra day, the statement `userAgeDays = userAgeDays + (userAgeYears / 4)` adds the number of leap years to userAgeDays. Note that the parentheses are not needed but are used to make the statement easier to read.

The program also computes and outputs the user's age in minutes.

Figure 10.9.2: Health data: Calculating user's age in days and minutes.

```c
#include <stdio.h>

int main(void) {
   int userAgeYears;
   int userAgeDays;
   int userAgeMinutes;

   printf("Enter your age in years: ");
   scanf("%d", &userAgeYears);

   userAgeDays = userAgeYears * 365;               // Calculate days without leap years
   userAgeDays = userAgeDays + (userAgeYears / 4); // Add days for leap years

   printf("You are %d days old.\n", userAgeDays);

   userAgeMinutes = userAgeDays * 24 * 60;         // 24 hours/day, 60 minutes/hour
   printf("You are %d minutes old.\n", userAgeMinutes);

   return 0;
}
```

```
Enter your age in years: 19
You are 6939 days old.
You are 9992160 minutes old.
```


PARTICIPATION ACTIVITY 10.9.2: Calculating user age in days.

1) The expression `(userAgeYears / 4)` assumes a leap year occurs every four years?
   - True
   - False

2) The statement `userAgeDays = userAgeDays + (userAgeYears / 4);` requires parentheses to evaluate correctly.
   - True
   - False

3) If the user enters 20, what is userAgeDays after the first assignment statement?
   - 7300
   - 7305

4) If the user enters 20, what is userAgeDays after the second

assignment statement?

- 7300
- 7305

Estimating total heartbeats in user's lifetime

The program is incrementally extended again to calculate the approximate number of times the user's heart has beat in his/her lifetime using an average heart rate of 72 beats per minutes.

Figure 10.9.3: Health data: Calculating total heartbeats lifetime.

```
#include <stdio.h>

int main(void) {
   int userAgeYears;
   int userAgeDays;
   int userAgeMinutes;
   int totalHeartbeats;
   int avgBeatsPerMinute = 72;

   printf("Enter your age in years: ");
   scanf("%d", &userAgeYears);

   userAgeDays = userAgeYears * 365;               // Calculate days without leap years
   userAgeDays = userAgeDays + (userAgeYears / 4); // Add days for leap years

   printf("You are %d days old.\n", userAgeDays);

   userAgeMinutes = userAgeDays * 24 * 60;         // 24 hours/day, 60 minutes/hour
   printf("You are %d minutes old.\n", userAgeMinutes);

   totalHeartbeats = userAgeMinutes * avgBeatsPerMinute;
   printf("Your heart has beat %d times.\n", totalHeartbeats);

   return 0;
}
```

```
Enter your age in years: 19
You are 6939 days old.
You are 9992160 minutes old.
Your heart has beat 719435520 times.
```


PARTICIPATION ACTIVITY 10.9.3: Calculating user's heartbeats.

1) Which variable is initialized when declared?

- userAgeYears
-

totalHeartbeats

- avgBeatsPerMinute

2) If the user enters 10, what value is held in totalHeartbeats after the statement `userAgeDays = userAgeYears * 365;`

- 3650
- 5258880
- Unknown

Limits on int values

In the above example, a userAge value of 57 or greater may yield an incorrect output for totalHeartbeats. The reason is that an int variable can typically only hold values up to about 2 billion; trying to store larger values results in "overflow". Other sections discuss overflow as well as other data types that can hold larger values.

10.10 Additional practice: Tweet decoder

This section has been set as optional by your instructor.

The following is a sample programming lab activity; not all classes using a zyBook require students to fully complete this activity. No auto-checking is performed. Users planning to fully complete this program may consider first developing their code in a separate programming environment.

The following program decodes a few common abbreviations in online communication as communications in Twitter ("tweets") or email, and provides the corresponding English phrase.

zyDE 10.10.1: Tweet decoder.

Load default template...

```
1 #include <stdio.h>
```

LOL

```
#include <string.h>

int main(void){
   char origTweet[10];

   printf("Enter abbreviation from t
   scanf("%s", origTweet);

   if (strcmp(origTweet, "LOL") == 0
      printf("LOL = laughing out lou
   }
   else if (strcmp(origTweet, "BFN")
      printf("BFN = bye for now\n");
   }
   else if (strcmp(origTweet, "FTW")
      printf("FTW = for the win\n");
   }
```

Run

Create different versions of the program that:

1. Expands the number of abbreviations that can be decoded. Add support for abbreviations you commonly use or search the Internet to find a list of common abbreviations.
2. For abbreviations that do not match the supported abbreviations, check for common misspellings. Provide a suggestion for correct abbreviation along with the decoded meaning. For example, if the user enters "LLO", your program can output "Did you mean LOL? LOL = laughing out loud".

10.11 Additional practice: Dice statistics

This section has been set as optional by your instructor.

The following is a sample programming lab activity; not all classes using a zyBook require students to fully complete this activity. No auto-checking is performed. Users planning to fully complete this program may consider first developing their code in a separate programming environment.

Analyzing dice rolls is a common example in understanding probability and statistics. The following calculates the number of times the sum of two dice (randomly rolled) equals six or seven.

zyDE 10.11.1: Dice rolls: Counting number of rolls that equals six or seven.

Load default templ

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void){
   int i;           // Loop counter iterates numRolls times
   int numRolls;    // User defined number of rolls
   int numSixes;    // Tracks number of 6s found
   int numSevens;   // Tracks number of 7s found
   int die1;        // Dice values
   int die2;        // Dice values
   int rollTotal;   // Sum of dice values

   numSixes = 0;
   numSevens = 0;

   printf("Enter number of rolls: \n");
   scanf("%d", &numRolls);
```


10

Run

Create different versions of the program that:

1. Calculates the number of times the sum of the randomly rolled dice equals each possible value from 2 to 12.
2. Repeatedly asks the user for the number of times to roll the dice, quitting only when the user-entered number is less than 1. Hint: Use a while loop that will execute as long as numRolls is greater than 1. Be sure to initialize numRolls correctly.
3. Prints a histogram in which the total number of times the dice rolls equals each possible value is displayed by printing a character like * that number of times, as shown below.

Figure 10.11.1: Histogram showing total number of dice rolls for each possible value.

```
Dice roll histogram:

2:  ******
3:  ****
4:  ***
5:  ********
6:  ******************
7:  ************
8:  ************
9:  **************
10: ***********
11: *****
12: ****
```


10.12 LAB*: Program: Online shopping cart (Part 1)

This section has been set as optional by your instructor.

(1) Create three files to submit:

- ItemToPurchase.h - Struct definition and related function declarations
- ItemToPurchase.c - Related function definitions
- main.c - main() function

Build the ItemToPurchase struct with the following specifications:

- Data members (3 pts)
  - char itemName [ ]
  - int itemPrice
  - int itemQuantity
- Related functions
  - MakeItemBlank() (2 pts)
    - Has a pointer to an ItemToPurchase parameter.
    - Sets item's name = "none", item's price = 0, item's quantity = 0
  - PrintItemCost()
    - Has an ItemToPurchase parameter.

Ex. of PrintItemCost() output:

```
Bottled Water 10 @ $1 = $10
```

(2) In main(), prompt the user for two items and create two objects of the ItemToPurchase struct. Before prompting for the second item, enter the following code to allow the user to input a new string. **c** is declared as a char. (2 pts)

```
c = getchar();
while(c != '\n' && c != EOF) {
   c = getchar();
}
```


Ex:

```
Item 1
Enter the item name:
Chocolate Chips
Enter the item price:
3
Enter the item quantity:
1

Item 2
Enter the item name:
Bottled Water
Enter the item price:
1
Enter the item quantity:
10
```

(3) Add the costs of the two items together and output the total cost. (2 pts)

Ex:

```
TOTAL COST
Chocolate Chips 1 @ $3 = $3
Bottled Water 10 @ $1 = $10

Total: $13
```


LAB

10.12.1: LAB*: Program: Online shopping cart (Part 1)

ACTIVITY 10.12.1: LAB : Program: Online shopping cart (Part 1)

0 / 9

Current file: main.c Load default template...

```
1  #include<stdio.h>
2  #include<string.h>
3
4  #include "ItemToPurchase.h"
5
6  int main(void) {
7
8     /* Type your code here */
9
10    return 0;
11 }
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.c (Your program) → Outp

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

10.13 LAB*: Program: Online shopping cart (Part 2)

This section has been set as optional by your instructor.

This program extends the earlier "Online shopping cart" program. (Consider first saving your earlier program).

(1) Extend the ItemToPurchase struct to contain a new data member. (2 pt)

- char itemDescription[ ] - set to "none" in MakeItemBlank()

Implement the following related functions for the ItemToPurchase struct.

- PrintItemDescription()
  - Has an ItemToPurchase parameter.

Ex. of PrintItemDescription() output:

```
Bottled Water: Deer Park, 12 oz.
```

(2) Create three new files:

- ShoppingCart.h - struct definition and related function declarations
- ShoppingCart.c - related function definitions
- main.c - main() function (Note: main()'s functionality differs from the warm up)

Build the ShoppingCart struct with the following data members and related functions. Note: Some can be function stubs (empty functions) initially, to be completed in later steps.

- Data members (3 pts)
  - char customerName [ ]
  - char currentDate [ ]
  - ItemToPurchase cartItems [ ] - has a maximum of 10 slots (can hold up to 10 items of any quantity)
  - int cartSize - the number of filled slots in array cartItems [ ] (number of items in cart of any quantity)
- Related functions
  - AddItem()

        - Adds an item to cartItems array. Has parameters ItemToPurchase and ShoppingCart. Returns ShoppingCart object.
    - RemoveItem()
        - Removes item from cartItems array (does not just set quantity to 0; removed item will not take up a slot in array). Has a char[ ](an item's name) and a ShoppingCart parameter. Returns ShoppingCart object.
        - If item name cannot be found, output this message: `Item not found in cart. Nothing removed.`
    - ModifyItem()
        - Modifies an item's description, price, and/or quantity. Has parameters ItemToPurchase and ShoppingCart. Returns ShoppingCart object.
    - GetNumItemsInCart() (2 pts)
        - Returns quantity of all items in cart. Has a ShoppingCart parameter.
    - GetCostOfCart() (2 pts)
        - Determines and returns the total cost of items in cart. Has a ShoppingCart parameter.
    - PrintTotal()
        - Outputs total of objects in cart. Has a ShoppingCart parameter.
        - If cart is empty, output this message: `SHOPPING CART IS EMPTY`
    - PrintDescriptions()
        - Outputs each item's description. Has a ShoppingCart parameter.

Ex. of PrintTotal() output:

```
John Doe's Shopping Cart - February 1, 2016
Number of Items: 8

Nike Romaleos 2 @ $189 = $378
Chocolate Chips 5 @ $3 = $15
Powerbeats 2 Headphones 1 @ $128 = $128

Total: $521
```

Ex. of PrintDescriptions() output:


```
John Doe's Shopping Cart - February 1, 2016

Item Descriptions
Nike Romaleos: Volt color, Weightlifting shoes
Chocolate Chips: Semi-sweet
Powerbeats Headphones: Bluetooth headphones
```

(3) In main(), prompt the user for a customer's name and today's date. Output the name and date. Create an object of type ShoppingCart. (1 pt)

Ex:

```
Enter Customer's Name:
John Doe
Enter Today's Date:
February 1, 2016

Customer Name: John Doe
Today's Date: February 1, 2016
```


(4) Implement the PrintMenu() function in main.c to print the following menu of options to manipulate the shopping cart. (1 pt)

Ex:

```
MENU
a - Add item to cart
r - Remove item from cart
c - Change item quantity
i - Output items' descriptions
o - Output shopping cart
q - Quit
```

(5) Implement the ExecuteMenu() function in main.c that takes 2 parameters: a character representing the user's choice and a shopping cart. ExecuteMenu() performs the menu options (described below) according to the user's choice, and returns the shopping cart. (1 pt)

(6) In main(), call PrintMenu() and prompt for the user's choice of menu options. Each option is represented by a single character.

If an invalid character is entered, continue to prompt for a valid choice. When a valid option is entered, execute the option by calling ExecuteMenu() and overwrite the shopping cart with the returned shopping cart. Then, print the menu and prompt for a new option. Continue until the user enters 'q'. *Hint: Implement Quit before implementing other options.* (1 pt)

Ex:

```
MENU
a - Add item to cart
r - Remove item from cart
c - Change item quantity
i - Output items' descriptions
o - Output shopping cart
q - Quit

Choose an option:
```


(7) Implement the "Output shopping cart" menu option in ExecuteMenu(). (3 pts)

Ex:

```
OUTPUT SHOPPING CART
John Doe's Shopping Cart - February 1, 2016
Number of Items: 8

Nike Romaleos 2 @ $189 = $378
Chocolate Chips 5 @ $3 = $15
Powerbeats Headphones 1 @ $128 = $128

Total: $521
```

(8) Implement the "Output item's description" menu option in ExecuteMenu(). (2 pts)

Ex:

```
OUTPUT ITEMS' DESCRIPTIONS
John Doe's Shopping Cart - February 1, 2016

Item Descriptions
Nike Romaleos: Volt color, Weightlifting shoes
Chocolate Chips: Semi-sweet
Powerbeats Headphones: Bluetooth headphones
```


(9) Implement "Add item to cart" menu option in ExecuteMenu(). (3 pts)

Ex:

```
ADD ITEM TO CART
Enter the item name:
Nike Romaleos
Enter the item description:
Volt color, Weightlifting shoes
Enter the item price:
189
Enter the item quantity:
2
```


(10) Implement the "Remove item from cart" menu option in ExecuteMenu(). (4 pts)

Ex:

```
REMOVE ITEM FROM CART
Enter name of item to remove:
Chocolate Chips
```

(11) Implement "Change item quantity" menu option in ExecuteMenu(). *Hint: Make new ItemToPurchase object before using ModifyItem() function.* (5 pts)

Ex:

```
CHANGE ITEM QUANTITY
Enter the item name:
Nike Romaleos
Enter the new quantity:
3
```


LAB ACTIVITY | 10.13.1: LAB*: Program: Online shopping cart (Part 2) | 0 / 30

Current file: **main.c** Load default template...

```
1 /* Type your code here */
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.c (Your program) → Outp

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

10.14 LAB: Simple car

This section has been set as optional by your instructor.

Given two integers that represent the miles to drive forward and the miles to drive in reverse as user inputs, create a SimpleCar variable that performs the following operations:

- Drives input number of miles forward
- Drives input number of miles in reverse

- Honks the horn
- Reports car status

SimpleCar.h contains the struct definition and related function declarations. SimpleCar.c contains related function definitions.

Ex: If the input is:

```
100 4
```


the output is:

```
beep beep
Car has driven: 96 miles
```


LAB ACTIVITY 10.14.1: LAB: Simple car 0 / 10

Current file: **main.c** Load default template...

```
1  #include <stdio.h>
2
3  #include "SimpleCar.h"
4
5  int main() {
6
7     /* Type your code here. */
8
9     return 0;
10 }
```

Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.c (Your program) → Outp

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

10.15 LAB: Vending machine

This section has been set as optional by your instructor.

Given two integers as user inputs that represent the number of drinks to buy and the number of bottles to restock, create a VendingMachine variable that performs the following operations:

- Purchases input number of drinks
- Restocks input number of bottles
- Reports inventory

VendingMachine.h contains the struct definition and related function declarations. VendingMachine.c contains related function definitions. A VendingMachine's initial inventory is 20 drinks.

Ex: If the input is:

```
5 2
```


the output is:

```
Inventory: 17 bottles
```


LAB ACTIVITY | 10.15.1: LAB: Vending machine 0 / 10

Current file: main.c Load default template...

```
1  #include <stdio.h>
2
3  #include "VendingMachine.h"
4
5  int main() {
6
7     /* Type your code here. */
8
9     return 0;
10 }
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.c (Your program) → Outp

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

10.16 LAB: Calculator

This section has been set as optional by your instructor.

Given main(), create the Calculator struct that emulates basic functions of a calculator: add, subtract, multiple, divide, and clear. The struct has one data member called **`value`** for the calculator's current value.

Implement the Calculator struct and related function declarations in Calculator.h, and implement the related function definitions in Calculator.c as listed below:

- InitCalculator(Calculator c) - initialize the data member to 0.0
- Calculator Add(double val, Calculator c) - add the parameter to the data member
- Calculator Subtract(double val, Calculator c) - subtract the parameter from the data member
- Calculator Multiply(double val, Calculator c) - multiply the data member by the parameter
- Calculator Divide(double val, Calculator c) - divide the data member by the parameter
- Calculator Clear(Calculator c) - set the data member to 0.0
- double GetValue(Calculator c) - return the data member

Given two double input values num1 and num2, the program outputs the following values:

1. The initial value of the data member, **`value`**
2. The value after adding num1
3. The value after multiplying by 3
4. The value after subtracting num2
5. The value after dividing by 2
6. The value after calling the clear() method

Ex: If the input is:

```
10.0 5.0
```

the output is:

```
0.0
10.0
30.0
25.0
12.5
0.0
```


LAB ACTIVITY | 10.16.1: LAB: Calculator | 0 / 10

File is marked as read only Current file: main.c

```
1  #include <stdio.h>
2
3  #include "Calculator.h"
4
5  int main() {
6     Calculator calc = InitCalculator();
7     double num1;
8     double num2;
9
10    scanf("%lf", &num1);
11    scanf("%lf", &num2);
12
13    // 1. The initial value
14    printf("%.1lf\n", GetValue(calc));
15
16    // 2. The value after adding num1
17    calc = Add(num1, calc);
18    printf("%.1lf\n", GetValue(calc));
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.c (Your program) → Outp

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

10.17 LAB: BankAccount struct

This section has been set as optional by your instructor.

Given main(), build a struct called BankAccount that manages checking and savings accounts. The struct has three data members: a customer name (string), the customer's savings account balance (double), and the customer's checking account balance (double). Assume customer name has a maximum length of 20.

Implement the BankAccount struct and related function declarations in BankAccount.h, and implement the related function definitions in BankAccount.c as listed below:

- BankAccount InitBankAccount(char* newName, double amt1, double amt2) - set the customer name to parameter newName, set the checking account balance to parameter amt1 and set the savings account balance to parameter amt2. (amt stands for amount)
- BankAccount SetName(char* newName, BankAccount account) - set the customer name
- void GetName(char* customerName, BankAccount account) - return the customer name in customerName
- BankAccount SetChecking(double amt, BankAccount account) - set the checking account balance to parameter amt
- double GetChecking(BankAccount account) - return the checking account balance
- BankAccount SetSavings(double amt, BankAccount account) - set the savings account balance to parameter amt
- double GetSavings(BankAccount account) - return the savings account balance
- BankAccount DepositChecking(double amt, BankAccount account) - add parameter amt to the checking account balance (only if positive)
- BankAccount DepositSavings(double amt, BankAccount account) - add parameter amt to the savings account balance (only if positive)
- BankAccount WithdrawChecking(double amt, BankAccount account) - subtract parameter amt from the checking account balance (only if positive)
- BankAccount WithdrawSavings(double amt, BankAccount account) - subtract parameter amt from the savings account balance (only if positive)
- BankAccount TransferToSavings(double amt, BankAccount account) - subtract parameter amt from the checking account balance and add to the savings account balance (only if positive)

LAB ACTIVITY 10.17.1: LAB: BankAccount struct 0 / 10

Current file: **BankAccount.h** Load default template...

```
1 /* TODO: Type your header file guards and include directives here. */
2
3
4 /* Type your code here. */
```

```
5
6
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **BankAccount.h** (Your program) → Ou

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

10.18 LAB: Product struct

This section has been set as optional by your instructor.

Given main(), build a struct called Product that will manage product inventory. Product struct has three data members: a product code (string), the product's price (double), and the number count of

product in inventory (int). Assume product code has a maximum length of 20.

Implement the Product struct and related function declarations in Product.h, and implement the related function definitions in Product.c as listed below:

- Product InitProduct(char *code, double price, int count) - set the data members using the three parameters
- Product SetCode(char *code, Product product) - set the product code (i.e. SKU234) to parameter code
- void GetCode(char *productCode, Product product) - return the product code in productCode
- Product SetPrice(double price, Product product) - set the price to parameter product
- double GetPrice(Product product) - return the price
- Product SetCount(int count, Product product) - set the number of items in inventory to parameter num
- int GetCount(Product product) - return the count
- Product AddInventory(int amt, Product product) - increase inventory by parameter amt
- Product SellInventory(int amt, Product product) - decrease inventory by parameter amt

Ex. If a new Product struct is created with code set to "Apple", price set to 0.40, and the count set to 3, the output is:

```
Name: Apple
Price: 0.40
Count: 3
```

Ex. If 10 apples are added to the Product struct's inventory, but then 5 are sold, the output is:

```
Name: Apple
Price: 0.40
Count: 8
```

Ex. If the Product struct's code is set to "Golden Delicious", price is set to 0.55, and count is set to 4, the output is:

```
Name: Golden Delicious
Price: 0.55
Count: 4
```


LAB ACTIVITY | 10.18.1: LAB: Product struct | 0 / 10

Current file: **Product.c** Load default template...

```
1 #include <stdio.h>
2 #include <string.h>
3
4 #include "Product.h"
5
6 /* Type your code here */
```

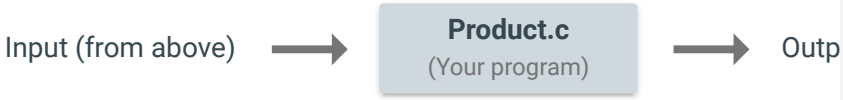

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program



Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

10.19 LAB: Student struct

This section has been set as optional by your instructor.

Given main(), build a struct called Student that represents a student that has two data members: the student's name (string) and the student's GPA (double). Assume student's name has a maximum length of 20 characters.

Implement the Student struct and related function declarations in Student.h, and implement the related function definitions in Student.c as listed below:

- Student InitStudent() - initializes name to "Louie" and gpa to 1.0
- Student SetName(char *name, Student s) - sets the student's name
- Student SetGPA(double gpa, Student s) - sets the student's GPA
- void GetName(char* studentName, Student s) - return the student's name in studentName
- double GetGPA(Student s) - returns the students GPA

Ex. If a new Student object is created, the default output is:

```
Louie/1.0
```

Ex. If the student's name is set to "Felix" and the GPA is set to 3.7, the output becomes:

```
Felix/3.7
```


LAB ACTIVITY 10.19.1: LAB: Student struct 0 / 10

File is marked as read only Current file: main.c

```
1  #include <stdio.h>
2  #include <string.h>
3
4  #include "Student.h"
5
6  int main() {
7      Student student = InitStudent();
8      char name[20];
9
10     GetName(name, student);
11     printf("%s/%.1lf\n", name, GetGPA(student));
12
13     student = SetName("Felix", student);
14     student = SetGPA(3.7, student);
15     GetName(name, student);
16     printf("%s/%.1lf\n", name, GetGPA(student));
17
18     return 0;
```


Develop mode | Submit mode Run your program as often as you'd like, before submitting

for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.c** (Your program) → Outp

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```